LECTURE : 4/20/05

Overview:

This lecture talked about Synchronization.

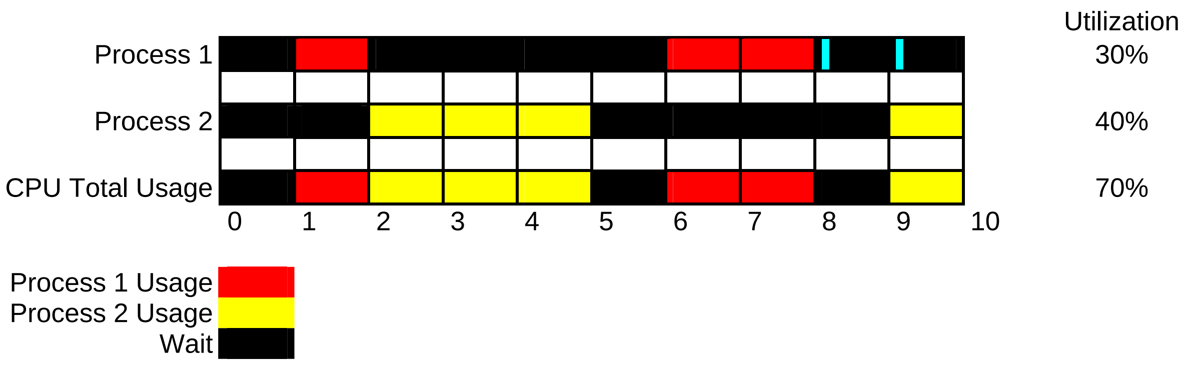
<u>Review:</u>
Why do we have processes?
Have the computer do more than one thing at a time.

<u>Feature of Processes</u>:

1. Isolation: each process has its own memory. The process does not interfere with other processes.
2. Efficiency: every process is either running (on the run queue) or block (on the wait queue).

What is the process waiting for?
Mostly the process is waiting for I/O (disk), or sometimes the network.

Since at most time the CPU is waiting for I/O, so if we only have one process, the utilization of the CPU is very low. In order to increase the usage of the CPU, we can have multiple processes. When one process is waiting, the other process can take chance to its jobs. In the following example, the utilization of the CPU is only 30% when there is only one process, but when there are two processes, the utilization can reach to 70%. Thus, the utilization of the CPU increases.



Threads: multiple processes share the same memory space.
As threads are sharing memory, threads only have efficiency, but not isolation. When programs want to have their own distinct privileges, we should use multiple processes. On the other hand, if some programs share their resources or need to exchange information, it is good to use multiple threads, since threads are easy for programs to communicate.

Programming Approach:

<u>**Goals** of this program</u>:
1. Read lines from 3 different files, and output them in any order to the screen.
2. Implement the program in the most significant ways.

<u>Synchronization Approach 1:</u>

```
#define NLINE 100000
#define LINESIZE 80
typedef char line_t[LINESIZE];
line_t lines[NLINES];
```

```
int main(int argc, char *argv[])
{
    int i;
    int lineno = 0;  //Pointer points to the line that is being read


    for(int i = 0; i < 3; i++){
        FILE *f = open(argv[i+1], "r"); //open the file
        if(f == NULL)
            abort();
                //read the file

//************* Problem of this program **************
/*This is code to read the data in one file. When the process starts reading the data, the whole process is blocked.
The process will not be end until reach the end of the file.
*/

        while(fgets(lines[lineno], LINESIZE, f) != NULL)
            lineno++;
        fclose(f);
    }

    for(i = 0; i < lineno; i++) //output all the lines
        fputs(line[i], stdout);
}
```

Problem of this program:
The above code works, but it is not efficient. If the process is blocked because the current file is not ready or something blocked in the current file, then this program does not have an implementation to jump to read the next file. This is a single process program. It blocks the machine when reading one file. It can't jump to the next file. In another word, it can't overlap the process. This program can reach Goal 1, which output the data in the files one line by one, but it can't reach Goal 2. For example, if the first file has 2 lines, the second file has 1 line, and the third file has 1 line. This program will read the 2 lines in the first file first, then the line in the second file, then the line in the third file. So the output of the program will be 1 1 2 3 (here 1 means any line in the first file.) What we want is create 3 processes or threads, one for each file. When the first process is reading the first line in the first file (it blocks the first process), the machine switches to the second process, and read the line in the second file. While both first and second processes are blocked by reading the file, the CPU switches to handle the third process which reads the third file. So the output will look like 1 2 3 1.

Thus our next goal is to overlap the reads from the file, so if one file blocks the program still has the ability to read from the other files.

Synchronization Approach 2:

Multiple Thread and Synchronization:
Overlap 3 threads, while 1 file is blocked, the CPU can switch to another thread and read another file.

```
int lineno;

void thread_run(void * thunk)
{
      FILE *f = (FILE *) thunk;

// blocking code taken from the non-threaded code
      while(fgets(lines[lineno], LINESIZE, f) != NULL)
            lineno++;
      fclose(f);

      pthread_exit(0);
}

int main(int argc, char *argv[])
{
      int i;
      void *thunk;
      pthread_attr_t thread_attr;
      pthread_t threads[3];

      pthtread_attr_init(&thread_attr);

      for(i = 0; i < 3; i++){
            FILE *f = fopen(argv[i + 1], "r");
            if(f == NULL
            || pthread_create(&threads[i], &thread_attr, threaad_run,f) < 0)

            /* pthread_create is a function used to create the threads and make the
            threads runnable, but that doesn't mean the thread is running immediately.
            */

                  abort();

      }

      //same as wait, wait for each thread done.
      for(i = 0; i < 3; i++)
            pthread_join(thrads[i], &thunk);

      //same as in the non-threaded code, output the lines
      for(i = 0; i < lineno, i++)
            fputs(lines[lineno], stdout);
}
```

How many threads do we want?

We want three threads in this example, one thread for each file that we are reading. There are a total of four threads in the program, since the main function is another thread. In general, we want one thread for each operation that can block the process.

Which are the block operations then?

read ( ), write ( ), open ( ), status ( ) …

How do we write the threaded code?

We can take the blocking code from the non-threaded code and put it in the thread function. (See the code above, it is the function thread_run).

<u>Problems of this program</u>:

The share variable lineno is unsafe. For example, if we have three files, the file has two lines (a a), the second file contains two line (b b), and the third file contains two lines (c c), then consider the following table. Are these possible outputs?

I	II	III	IV
a	c	c	c
a	n/a (no new lines)		
b	n/a		
b	n/a	c	c
c	n/a		n/a
c	n/a		n/a

I is possible, this is what we want.

II is not possible, since each thread will read two characters, and stick them into lines[ ], but both characters cannot be placed on the same line.

III is possible. Three processes start up when lineno = 0. The first process reads first, and the CPU switches to the second process. When the third process reads the character and sticks it into lines[lineno], the lineno has not been increased by any process, so at this time the lineno is 0, and “c” overwrites the previous output in lines[0] as it is the last process to perform the read. Therefore, “c” appears in the first line. Then all processes increase the lineno by one. So at this time the lineno = 3. Same procedure happens again, the output of the third process overwrites the previous outputs, and hence “c” appears on the fourth line. Then each process increases the lineno, so two more new lines were printed.

IV is possible. When the lineno = 0, we assume the same procedure happened as above. Now lineno = 3. In assembly the line lineno ++ is equivalent to the following three lines of code:

```
movl     lineo     %eax           // 1
incl     %eax                     // 2
movl     %eax    lineno           // 3
```

When the program reaches the line lineno++, then the machine performs the three above lines. Now as a programmer we assume that each process would perform the three lines at different times and hope the increment is done in the correct order, but that is not true. Now assume that we are done with the printing of the first line of each of the three process, and have only ‘c’ as the output on line number 0 and then three blank lines. (Reason: because at the beginning before read each file, all files get the lineno = 0, so the data reading from the last file will overwrite the data in the previous two files) So, now the lineno variable is 3. Right now all three files are ready to increase the lineno by 1. Supposing that all three threads interrupted when they finish the first instruction of the Assembly code above, then the lineno for all three threads is 3. In that case, after all three threads increase the lineno, lineno = 3 + 1 = 4; Thus, the lineno is the same for all three threads. The data in the last read file overwrites the data in the previous files, since they have the same lineno, which means all the data goes into the same memory location. Thus we have the output in option IV, and the lineno = 4.

The problem here is called “Race Condition” (“Heisenbug”). We as a programmer want that the line lineno++ be performed by a single process at any given time, so if process ‘a’ is performing the increment, then process ‘b’ should not be able to interfere with the performance of process ‘a’ until ‘a’ is done. We want the program to execute in the following order a1 a2 a3, b1 b2 b3, c1 c2 c3. But in the above code the OS can perform the execution in any order, like a1 b1 c1 a2 a3 b2 b3 c2 c3. What we need is a1 a2 a3 always happen in the same block. Thus we call such blocks of code as “critical section”.

Critical Section is a block of code that has atomic execution, which means that no interrupt can happen inside of this block. In order to have high efficiency, we want the critical section to be as small as possible. Usually, we only put the non-blocking code (the code that can run fast) in the critical section.

How to implement critical section?

The way to implement critical section is using lock with the following codes:

Before the critical section obtain the lock:

acquire_lock(&lock);

After the critical section release the lock:

release_lock(&lock);

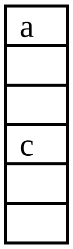
The details of acquiring and releasing the locks are discussed in details at the end of the next synchronization approach.

In Approach 2, it is possible to interrupt a thread after it has acquired the value of lineno, and before it finishes incrementing lineno. This resulted in the problematic code as shown above. Now we have another approach. Here we use the idea of critical section to avoid the race condition issue. The code below is an example of what code should be implemented as critical section. Well from our discussion above lineno++ sure should be a critical section code, thus the coloring for the line (indication of critical section).

```
void thread_run(void * thunk)
{
      FILE *f = (FILE *) thunk;
      while(fgets(lines[lineno], LINESIZE, f) != NULL)
            lineno++;
      fclose(f);

      pthread_exit(0);
}
```

There still is a problem with the code with the above implementation, If we make the critical section like the one above then we still may get the following output:



As the lineno can start at 0, and all threads get 0 as the starting line, thus there is nothing to stops all the threads from using the value 0 as a start value. Thus, there need to be more changes in the code, and as suggested by a student in class, we have the following critical section:

```
void thread_run(void * thunk)
{
      FILE *f = (FILE *) thunk;

      while(1){
            if (fgets(lines[lineno], LINESIZE, f) != NULL)
                  break;
            lineno++;
      }
      fclose(f);
```

```
	pthread_exit(0);
}
```

Problem: This approach contains the block operation in the critical section, so it is the same as the non-threaded program. The code above is inefficient. It works but may block after every line, defeating the purpose of threads.

Another solution, increase the line before reading:

```
void thread_run(void * thunk)
{
	FILE *f = (FILE *) thunk;
	while(1){
		int my_line = lineno;
		lineno++;
		if (fgets(lines[lineno], LINESIZE, f) != NULL)
			break;
	}
	fclose(f);
	pthread_exit(0);
}
```

Problem: The above code is more efficient and works. The slight problem with the code is that it would guarantee to get some blank lines, as we reserve space for a line without knowing there is something to be read from the file.

Synchronization Approach 3:

```
int lineno;

void thread_run(void *thunk)
{
	FILE *f = (FILE *) thunk;
	while(l){
		int my_lineno;

//critical section:
		acquire_lock();
		my_lineno = lineno;
		lineno++;
		realease_lock();
		if(fgets(lines[my_lineno], LINESIZE, f) == NULL){
			fclose(f);
			pthread_exit(void *) my_lineno;
		//here return the empty lineno to the main function.
		}
	}
}

int main(int argc, char *argv[])
{
	int i, empty_linenos[3];
	void *thunk;
	pthread_attr_t thread_attr; pthread_t threads[3];
	pthread_attr_init(&thread_attr);
	for(i = 0; i < 3; i++){
		FILE *f = fopen(argv[i+1], "r");
		if(f == NULL || pthread_create(&threads[i], &thread_attr, thread_run, f) <0)
			abort();
	}
/*get the return value of blank lineno from thread_run, and store in the array
empty_linenos. */
      for(i = 0; i < 3; i++){
            pthread_join(threads[i], &thunk);
            empty_linenos[i]= (int) thunk;
      }

/*At this point all threads have joined; we are the only thread running. Fill in the
“holes” replace the empty line with the line at the end of the file, and decrease the
lineno by 1. */

      for(i = 0; i < 3; i++){
            memcpy(lines[empty_lineno[i]], lines[lineno-1], LINESIZE);
            lineno--;
      }
      for(i = 0; i < lineno;i++)
            fputs(lines[lineno], stdout);
}
```

In the example above, we increase the value of lineno before we read the file, so we increment the number of lines to be read before reading the file. Since we have three files to read, the lineno is increased by 1 by each file process, so when the all the work has been done, it will produce 3 extra empty lines on the screen. Now, we have another issue to solve, we need to get rid of the empty lines before we output to the screen.

One approach to get rid of the empty lines is to use a temporary buffer to hold each line, if the buffer does not have empty line, then copy it to the global buffer, lines[]; otherwise, don’t copy. The code for this solution is:

```
char *linebuf[];
while(fgets(linebuf[my_lineno]. LINESIZE, f) != NULL) {
      memcpy(lines[lineno], linebuf[0]);
      //the following is the same as above
}
```

The above solution gets rid of the empty lines, but it requires copying each line from the temporary buffer to lines array. It doubles the work.

Another solution as shown in the synchronization approach 3 is to use an array empty_lineno to hold the number of empty lines. After finishing reading all the files, look into the empty_lineno to find out the empty line in the buffer then replace the empty line with the non-empty line from the end of the buffer and decrease the lineno by one. Since we can output the data in any order, this is a good solution to get rid of the empty line and reduce the work.

Properties for Critical Section:

Property 1: Mutual execution: that acquire_lock will hold until *no other thread* in between acquire_lock and release_lock on the same lock.

Does acquire lock produce a good critical section if only the above property?
Look at the following example:

```
acquire_lock(l) {
      while(1)
            /* do something */;
}
```

This code can satisfy the first property of acquire lock, but it can't get out of the lock, because of infinite loop. Thus we need more properties for the lock to produce a good critical section.

Property 2: Progress: means that if anyone is at acquire_lock(); *someone* will eventually succeed.

Again, does property 1 and property 2 together is good enough for a lock?
Here is another example:

Process A:

```
while(1){
        acquire_lock();
        printf("a");
        release_lock();
}
```

Process B:

```
while(1){
acquire_lock();
        printf("b");
        release_lock();
}
```

The example code above satisfies mutual exclusion and progress, but it still has a problem. When these two processes put together to execute, it is possible that one process executes all the time, but the other process does not get a chance to execute, thus starvation. Usually the CPU will switch from one process to the other process if there is an interrupt. In order to get a critical section, we need to disable interrupts once the process has acquired the lock. So, if process A gets the lock and never releases the lock then process B could starve, and thus to avoid processes from starvation we need another property.

Property 3: Fairness / no starvation, so that no one waits forever, that is, every process in the wait queue would get a chance to execute.

Right now, acquire lock can produce a good critical section, then how should we implement it to use? Think about this: what might cause something to run? The answer is interrupting (context switches). Thus we want to produce critical sections in the kernel when interrupt happens:

```
        acquire_lock();
                // turn of the interrupts
        release_lock();
                // turn on the interrupts
```

SPINLOCK example:

```
Acquire(int *l){
        while(1) {
                turn off interrupt;

        /*atomic execution to make sure no interrupt to change the value of variable 'l' in the middle */
                if(*l = = 0){
                        *l = 1;
                        turn on interrupt;
                        return;
                }
                turn on interrupt
}
release(int *l)
        *l = 0;
```

The above code does not guarantee fairness and starvation. The solution to produce fairness:

```
  Acquire(int *l){
        while(1) {
              turn off interrupt;

        /*atomic execution to make sure no interrupt to change the value of variable l in  the middle */
              if(*l = = 0){
                    *l = 1;
                    turn on interrupt;
                    return;
              }
              turn on interrupt
              //************* solution for fairness ***************
              //if the process can not get a chance to execute, wait until it gets a chance.
}
release(int *l)
        *l = 0;
```

At the end of the class, we bring up the idea of **Event loop**.

```
int main(int argc, char *argv[])
{
      int i, lineno = 0;
      fd_set fds;
      FILE *fs[3];

      for(i = 0; i < 3; i++){
            fs[i] = fopen(argv[i+ 1], "r");
            if(fs[i] == NULL)
                  abort();
            fcntl(fileno(fs[i], F_SETEL, O_NONBLOCK);
      }
//Event loop:

      while(fs[0] || fs[1] || fs[2]) { //at least one file is open, do non-block at all
open files, BLOCK on the UNION of all the files
            for(i = 0; i < 3; i++)
                  if(fs[i] != NULL) {
                        if(fgets(lines[lineno], LINESIZE, fs[i] != NULL)
                              lineno++;
                        else if(errno != EAGAIN) {
                              fclose(fs[i]);
                              fs[i] = NULL;
                        }
                  }

                  FD_ZERO(&fs);
                  for(i = 0; i < 3; i++)
                        if(fs[i] != NULL)
                              FD_SET(&fds, fileno(fs[i]));
                  select(MAXFD, &fds, NULL, NULL, NULL);
      }
      for(i = 0; i < lineno; i++)
            fputs(lines[lineno], stdout);
}
```