Announcement

- We have studied and used POSIX timers for periodic tasks before.
- Today: using threads
  - data sharing using mutex
  - message passing
- To learn more on threads, “Programming with POSIX Threads” by David R. Butenhof, Addison-Wesley. Handouts are in the lab.
- To learn more on message queues, “Programming for The Real World, POSIX.4” by Bill O. Gallmeister, O’Reilly & Associates, Inc.

Processes and Threads

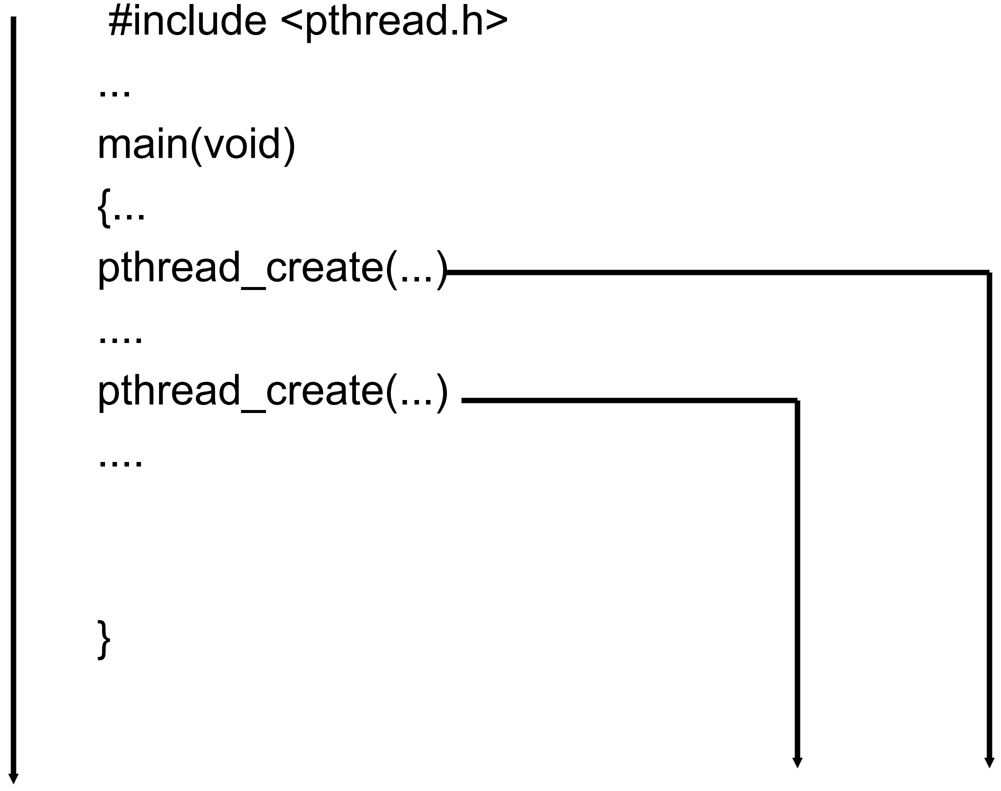
- In many applications, you need to keep many things active at the same time.
  - Processes have different address spaces and has a least 1 thread
  - Each process can have multiple threads sharing the the same address spaces
  - Threads in the same process can share variables protected by mutex (they can also use messages to communicate).
  - Threads in different processes can communicate by sending messages.
- You will learn how to do it now, and practice it in the Lab

Class Discussion

- Giving an application system that requires great robustness, what should be the guideline in using threads?
- Should we use threads at all?
- If we do, when and why? Give an example to illustrate your point.

Thread: concurrency within a process

Processes have separate address spaces but threads share the same address space.
Process offers better __________ while threads offers higher _________.



```
 #include <pthread.h>
...
main(void)
{...
pthread_create(...)
....
pthread_create(...)
....


}
```

All the threads will be killed when the main thread exits.

Many detailed thread management functions exist

Example Thread Applications

- In Lab 2, your program reads the high and low voltages and the frequency, you then display the wave form. What if users want to adjust the waveform specs from time to time without existing the program?

- what is wrong with embedding the read statements in the waveform loop?

- We need
  - thread 1: user interface thread
  - thread 2: wave form thread

- These two threads can communicate using shared variables or messages

- By the way, which thread should have higher priority and why?

Thread Creation

```
#include <pthread.h>
#include<resource.h>
void *wave_display(void* args)
{//get waveform spec and display the periodic wave form on the scope}

main(void)
{//get initial waveform spec from users
status = setpriority(PRIO_PROCESS, 0,  10); //set the process (main thread) priority to 10

pthread_t          wave_thread_id;
pthread_attr_t   wave_thread_attr;
status = pthread_attr_create(&wave_thread_attr); //*what does this line do?
//in current configuration, priority >= 17 won't be paged out.
wave_thread_attr.pthread_attr_inheritscheded = 0; //don't inherit creator's scheduling parameters
wave_thread_attr.pthread_attr_prio = 17;

status = pthread_create(&wave_thead_id, wave_thread_attr,  wave_display, NULL);

// communication loop with users, "do you want to change waveform spec ? ....."
}
```

*Ans: allocate memory to thread_attr

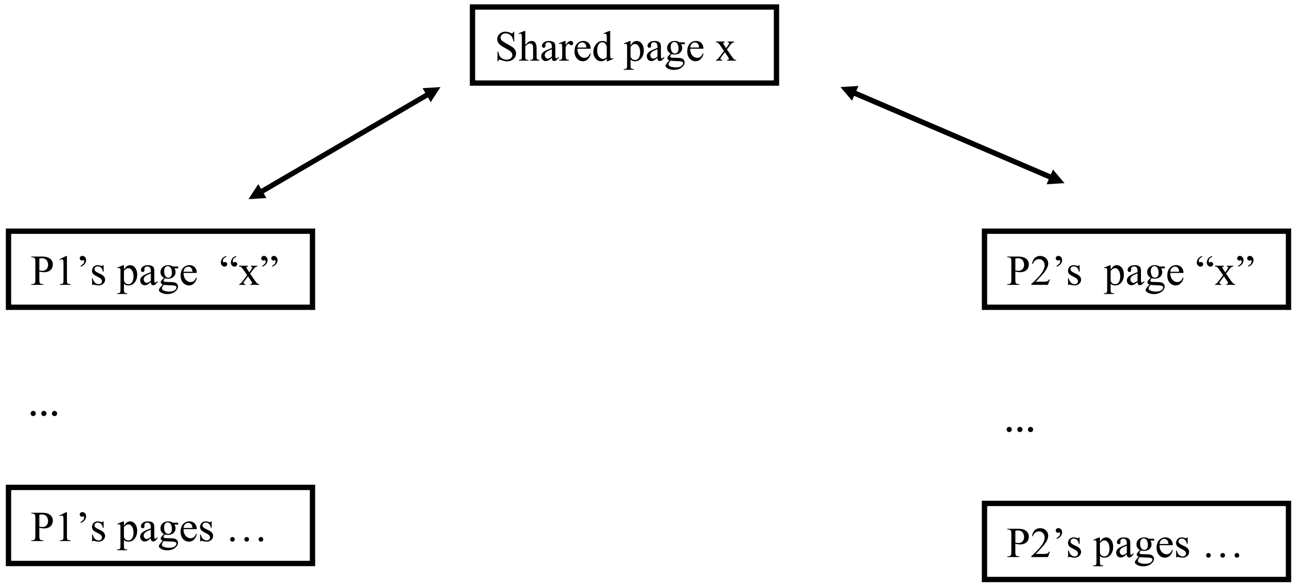
Using Shared Variables with Mutex

```
#include <pthread.h>
//declare and create waveform spec data structure
pthread_mutex_t  wave_spec_mutex

void *wave_display(void* args) {
  while(1) { //wave form loop
   status = pthread_mutex_lock(&wave_spec_mutex);
        //read waveform spec
   status = pthread_mutex_unlock(&wave_spec_mutex);
   //wait for posix timer signal,
   send out the voltage to display the waveform
  }  }
main(void)
{…
pthread_mutex_init(&wave_spec_mutex, NULL); //initialize mutex
while(1) { //communication loop
  //read in new wave spec
  status = pthread_mutex_lock(&wave_spec_mutex);
  //update waveform spec //keep critical section SHORT. Don't interact with user here!
  status = pthread_mutex_unlock(&wave_spec_mutex);
 }   }
```

Share Memory Region - 1

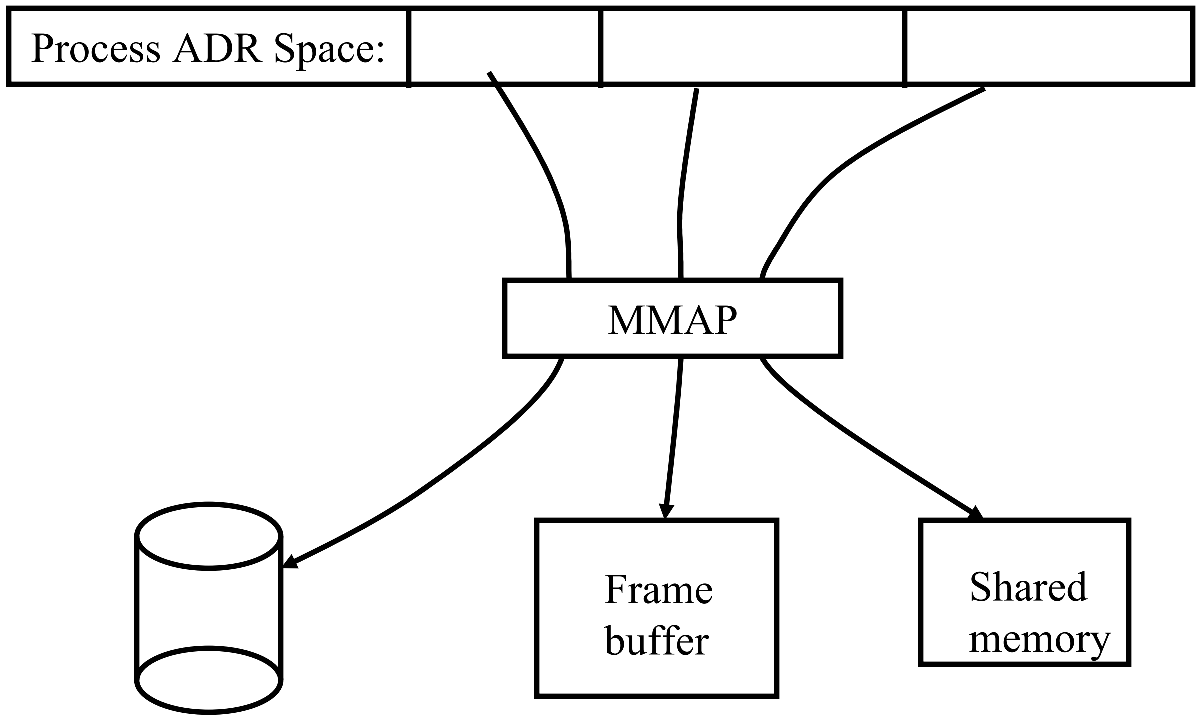
- By default, processes have different memory spaces. However, it is possible to let multiple processes to have a shared global memory region for efficient communications.
- Once the shared memory region is mapped into a process address space. Any read and write to that mapped space will be redirected to the shared region.



Using Shared Memory as a Debugging Tool

- The use of shared memory between process creates global variables that tightly couple processes. It is often an expedience that you will regret later as system grow and being modified.
- Use it with great care and have sufficient justification.
- An good and justified use in real time system is for debugging.
  - Suppose that you have a process that behaves strangely or crashes
  - If you use gdb, it could behave even more strange, since gdb badly distorts the timing behavior
  - You could create a ring buffer in shared memory, write important data to it, and peek at it via another monitoring process.
  - You can look at in the ring buffer even if the process being monitored crashed

MMap



Shared Memory Region - 2

- Create shared memory:
- #include <sys/nman.h>
- Int shm_open(const char *name, int oflag, mode_t mode);

e.g. proc_exec allows executing instructions there

- Setting memory size
- #include <unistd.h>
- Int ftruncate(int fd, off_t total_size);

- Mapping shared memory into my address
- Nbytes of shared memory starting at offset at shared memory is mapped to nbytes starting at *mem_location at your memory space.

- #include<sys/nman.h>
- Void *nmap(void *mem_location, size_t nbytes, int memory_protection,
-                         int mapping_flag, int fd, off_t offset);

                          // mem_location = 0 means let the system to decide
- When you are done
- #include <sys/nman.h>
- Int munmap(void *offset, size_t nbytes);

Mutex and System Reliability

- When you use shared memory, use mutex to ensure mutual exclusion when threads at different process read and write into the shared memory location.
- When processes and/or threads use mutex to communicate, it is possible that one thread locks the mutex and then fails, the system may hang as a result.
- Thus, it is preferable to use message passing across process boundaries.
- However, the use of mutex across application processor boundaries are sometimes unavoidable. For example, in a multi-processor with a shared memory board, any process that wants to do a bus transaction to get a block of shared memory, the bus must be locked during the transaction.
- How can we avoid the multi-processor being locked up, should one of the application process fails?

Using Message Queues

- Within a process, shared variables with mutex is an efficient way to communicate
- What about threads in different processes? Threads between different processes can communicate with message queues.

```
//user interface, create a message queue and sent waveform spec
void main(void)
{...
long buffer[4];
memset(buffer, 0, sizeof(buffer)); //clean up the buffer space

//buffer[0] for msg seq #, buffer[1] for min_volt, buffer[2] for max_volt, buffer[3] for freq

struct mq_attr    wave_q_attr
//set up the message queue attributes

memset(&wave_q_attr, 0, sizeof(wave_q_attr));//clear up any default settings
wave_q_attr.mq_flags = O_NONBLOCK; wave_q_attr.mq_maxmsg = 10;
wave_q_attr.mq_msgsize = sizeof(buffer);

...
(to be continued)
```

Sending Messages

```
(continue)
//WAVE_SPEC_Q  is our name for the message queue. It is GLOBAL across all processes.
mqd_t  wave_q_id;
wave_q_id = mq_open("WAVE_SPEC_Q", O_WR_ONLY | O_CREAT | NON_BLOCK, 0666,
       &wave_q_attr);
.//666    queue permission, same as file permission:  others(rwx), group(rwx), owner(rwx)..
while(command != 'q' ) {
 //ask user if (s)he wants to give new spec or quit
 // if 'q' is received, set the message sequence number to -1, otherwise, increment it by 1
  //get new waveform spec data from user and put the waveform spec  into the buffer
  buffer_length = sizeof(buffer);
  status = mq_send(wave_q_id, (char *) buffer, buffer_length, prio);
  //prio = the priority assigned to msg,  a rule of thumb is to use the thread priority.
  //Real-time messages are prioritized. Not FIFO.  Same priority msg are kept in FIFO order
}

 //clean up
  status = mq_close(wave_q_id);    //like closing a file, the message queue STILL lives on in the system
  status = mq_unlink(wave_q_id);   //destroy the message queue after everyone closes the queue
}
```

What will happen if you do not unlink??

Receiving A Message

```
//program 2: create a receiving queue for waveform spec msg, and display the wave form
void main(void)
{       long buffer[4];
        struct mq_attr    wave_q_attr
        //set up the  same message queue attribute
        memset(&wave_q_attr, 0, sizeof(wave_q_attr));
        wave_q_attr.mq_flags = O_NONBLOCK; wave_q_attr.mq_maxmsg = 10;
        wave_q_attr.mq_msgsize = sizeof(buffer);
        ...
        //set  up the buffer to store the waveform spec,
        //buffer[0] = msg seq #, buffer[1] = min_volt, buffer[2] = max_volt, buffer[3] = freq
        memset(buffer, 0, sizeof(buffer)); //clean up the buffer space
        mqd_t  wave_q_id;
        wave_q_id = mq_open("WAVE_SPEC_Q", O_RDONLY|O_CREAT|O_NONBLOCK,
                                0666, &wave_q_attr);
        while (msg_seq_no != -1) {
             buffer_length = sizeof(buffer);
             status = mq_receive(wave_q_id, (char *) buffer, buffer_length, &prior);
             //& prior: priority of the message that you just read.
             //check message sequence number, get the wave form spec from buffer
             //set up the posix timer and send out the voltages to display the waveform
        }      }
```

State of Message Queues

- Life span of a message queue
  - Once a message queue is created, like a file, its life is longer than the life of the program that creates it, unless it is unlinked.
  - Unlike a file, it will disappear if the system reboots
  - This feature permits debugging after a program fails and allows for different programs to read and write to shared message queues.

- In our previous example, what would happen if we use cntr_c, instead of using the “q”, to terminate the program, and then restart the program again?

```
//when WAVE_SPEC_Q becomes a zombie…
//you may run a zombie killer to clean up the mess
        ...
        wave_q_id = mq_open(“WAVE_SPEC_Q”, ....);
        status = mq_close(wave_q_id);
        status = mq_unlink(wave_q_id);
          ...
```

Finally, you may set up a notification event when the message queue changes from empty to not empty (See pp 105 to 106 in the handout)