DaVinci / OMAP Software Design Workshop

Multi-Thread System

- Linux Processes
- Linux Threads
- Thread Synchronization
- Using Real-Time Threads

What is Our Main Goal?

- **Goal:** Run video and audio at the same time
- **What are different ways to accomplish this?**

① Processor 1 (Mem): `main(){ funcAud(); … }` — Processor 2 (Mem): `main(){ funcVid(); … }`

Two processors
- Lots of real estate and memory, costly
- How do you sync A/V ?

② Process A (Mem): `main(){ funcAud(); … }` — Process B (Mem): `main(){ funcVid(); … }`

Two processes (cmd line)
- Cmd line: `./AUD.Beagle &` `./VID.Beagle`
- How do you sync A/V ?

③ **Two processes (programatic)**
- Start 2nd process programatically
- Memory protection (MMU)
- Context switch is difficult.
- How do you sync A/V ?

④ Process A (Mem): T1 A, T2 V

Let's compare/ contrast each of these…

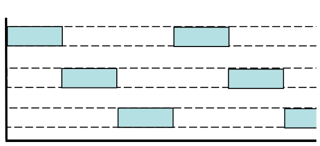
Two pThreads
- Uses fewer resources, faster
- pThreads (lightweight threads)
- Can share global variables

What is a Processor?

System #1

Processor A	Processor B	Processor C
Memory	Memory	Memory
`main(){ func1(); func2(); …`	`main(){ func3(); func4(); …`	`main(){ func5(); func6(); …`

What is a Process?

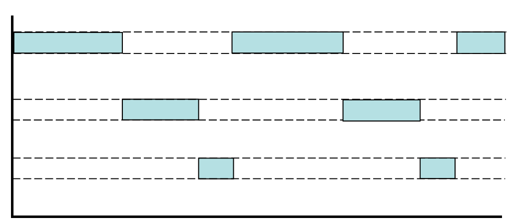


System #2

Processor A

Process A	Process B	Process C
Memory	Memory	Memory
`main(){ func1(); func2(); …`	`main(){ func3(); func4(); …`	`main(){ func5(); func6(); …`

Linux Time-Slice Scheduler



- **Processes are time-sliced with more time given to lower *niceness***
- **Linux dynamically modifies processes' time slice according to process behavior**
- **Processes which block are rewarded with greater percentage of time slice total**

Scheduling Methodologies

Time-Slicing with Blocking

Scheduler shares processor run time between all threads with greater time for higher priority

- ✓ No threads completely starve
- ✓ Corrects for non-"good citizen" threads
- × Can't guarantee processor cycles even to highest priority threads.
- × More context switching overhead

Linux Default

Thread Blocking Only

Lower priority threads won't run unless higher priority threads block (i.e. pause)

- × Requires "good citizen" threads
- × Low priority threads may starve
- ✓ Lower priority threads never break high priority threads
- ✓ Lower context-switch overhead

BIOS

Notes:
- Linux threads provide extensions for real-time thread behavior as well; however, time-slicing is the default
- Similarly, you can setup BIOS to time-slice threads (TSK's), but this is not the default for BIOS (i.e. real-time) systems

The Usefulness of Processes

Option 1: Audio and Video in a single Process

```
// audio_video.c
// handles audio and video in
//    a single application

int main(int argc, char *argv[])
{
    while(condition == TRUE){
        callAudioFxn();
        callVideoFxn();
    }
}
```

Option 2: Audio and Video in separate Processes

```
// audio.c, handles audio only

int main(int argc, char *argv[]) {
    while(condition == TRUE)
        callAudioFxn();
}
```

```
// video.c, handles video only

int main(int argc, char *argv[]) {
    while(condition == TRUE)
        callVideoFxn();
}
```

Splitting into two processes is helpful if:
1. audio and video occur at different rates
2. audio and video should be prioritized differently
3. multiple channels of audio or video might be required (modularity)
4. memory protection between audio and video is desired

Terminal Commands for Processes

# **ps**	Lists currently running user processes
# **ps -e**	Lists all processes
# **top**	Ranks processes in order of CPU usage
# **kill** *<pid>*	Ends a running process
# **renice** +5 –p *<pid>*	Changes time-slice ranking of a process (range +/- 20)

Launching a Process – Terminal

```
$ ./app_AUDIO &
[1] 15981
$
Debug: Requesting 12000 frame input buffer
…
$ ps
  PID TTY          TIME CMD
 1398 pts/0    00:00:02 bash
15981 pts/0    00:00:00 app_AUDIO
15985 pts/0    00:00:00 ps
$ kill 15981
$ ps
  PID TTY          TIME CMD
 1398 pts/0    00:00:02 bash
15993 pts/0    00:00:00 ps
[1]+  Terminated
```

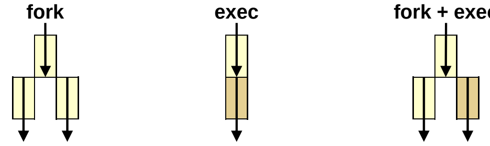
Launching a Process – Terminal

```
$ ./app_AUDIO &
[1] 15998
$
Debug: Requesting 12000 frame input buffer
…
$ jobs
[1]+  Running                 ./app_AUDIO &
$ kill %1
$ ps
  PID TTY          TIME CMD
 1398 pts/0    00:00:02 bash
16185 pts/0    00:00:00 ps
[1]+  Terminated              ./app_AUDIO
$ jobs
```

Side Topic – Creating New Processes in C

We won't actually need this for our lab exercises, though, we found it interesting enough to include it here.

fork **exec** **fork + exec**



Splits one executing process into two with same environment

New process replaces old but maintains previous environment

Launch new process and keep previous process

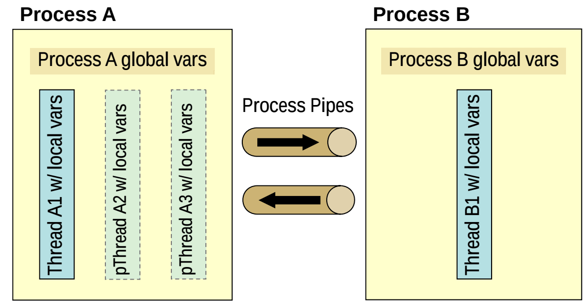
- All processes are *split-off* from the original process created at startup
- When using fork, both processes run the same code; to prevent this, test if newly created process and run another program – or exec to another program
- To review, a *process* consists of:
  - Context (memory space, file descriptors)
  - One (or more) threads

One *or more* threads per process?

Multi-Thread System

- Linux Processes
- Linux Threads
- Thread Synchronization
- Using Real-Time Threads
- Lab 08

Processes and Threads

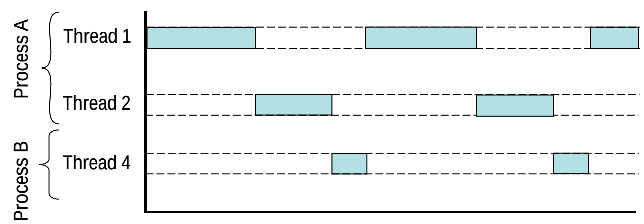


- By default, each process contains one main thread of execution
  - Additional threads can be spawned within a process (pThreads)
  - All threads within a process share global variables
- Threads scheduled individually by priority – regardless of which process they reside within
- No thread isolation – a rogue pointer will probably bring down all threads in that process.

Threads vs Processes

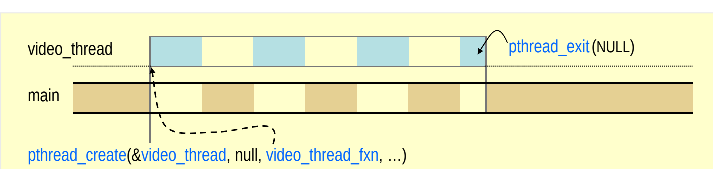
	Processes	Threads
Memory protection	✓	✗
Ease of use	✓	✗
Start-up cycles	✗	✓
Context switch	✗	✓
Shared globals	no	yes
Environment	program	function

Linux Scheduler



- **Entry point at main() for each process is scheduled as a thread.**
- **Threads are scheduled with time slicing and blocking as previously discussed for processes.**
- **Processes may then add additional threads to be scheduled.**

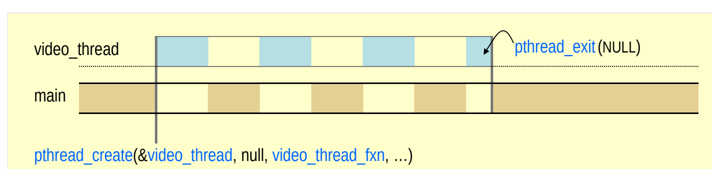
pThread Functions – Create & Exit

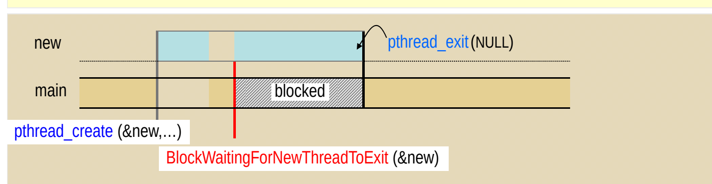


- Use pthread_create() to kickoff a new thread (i.e. child)
  - Starts new thread executing in the same process as its parent
  - As shown, both threads now compete for time from the Linux scheduler
  - Two important arguments – thread object, function to start running upon creation
- pthread_exit() causes child thread end
  - If _create's starting function exits, pthread_exit() is called implicitly

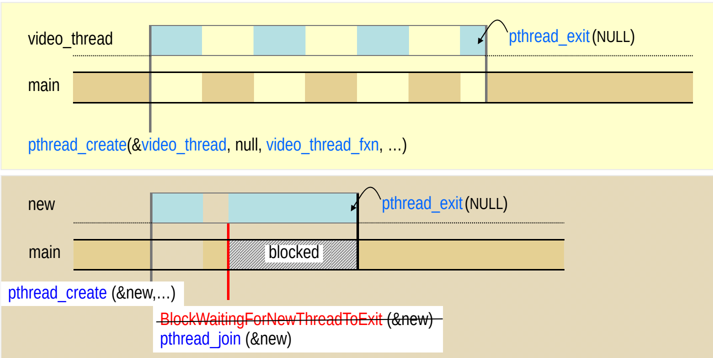
What if there's nothing for main() to do?

Waiting for the Child to Exit

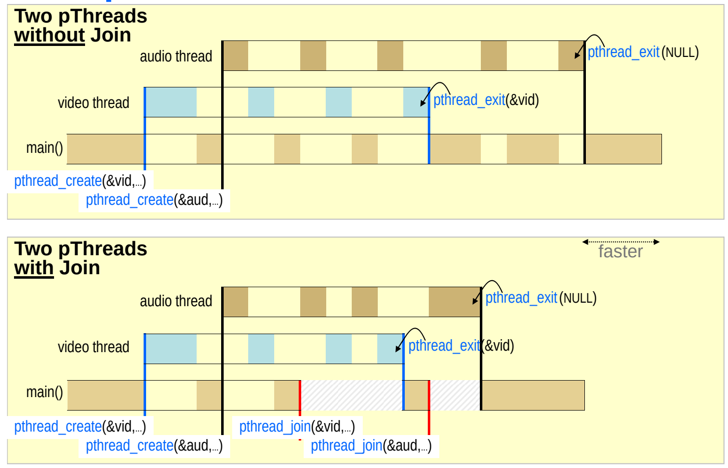




Re-Joining Main



Multiple Threads … With or Without Join



Multi-Thread System

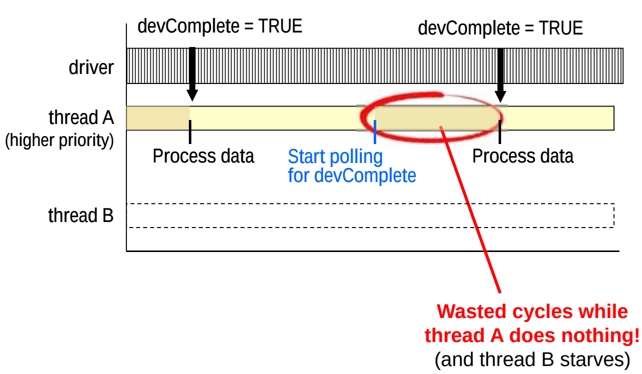
- Linux Processes
- Linux Threads
- **Thread Synchronization**
- Using Real-Time Threads
- Lab 08

Thread Synchronization (Polling)

```
void *threadA(void *env){
int test;
while(1) {
     while(test != TRUE) {
          test = (volatile int) env->driverComplete;     } Polling Loop
     }
     doSomething(env->bufferPtr);
}
```

- **Thread A's doSomething( ) function should only run after the driver completes reading in a new buffer**
- **Polling can be used to halt the thread in a spin loop until the driverComplete flag is thrown.**
- **But <u>polling is inefficient</u> because it wastes CPU cycles while the thread does nothing.**

Thread Synchronization (Polling)

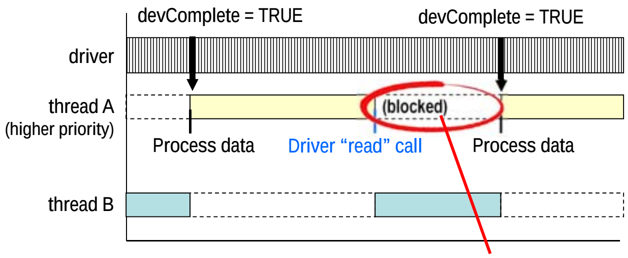


Thread Synchronization (Blocking)

```
void *threadA(void *env){
while(1){
     read(env->audioFd, env->bufferPtr, env->bufsize);   } Blocks (waits till complete)
     doSomethingNext(env->bufferPtr);
}
```

- **Instead of polling on a flag, the thread blocks execution as a result of the driver's read call**
- **More efficient than polling because thread A doesn't waste cycles waiting on the driver to fill the buffer**

Thread Synchronization (Blocking)



Thread blocks until driver fills buffer.
No wasted cycles!
(thread B gets to fill time)

- Semaphores are used to block a thread's execution until occurrence of an event or freeing of a resource
- Much more efficient system

Synchronization with Peripherals

ALSA driver: readi function is blocking
writei function blocks if outgoing buffer full

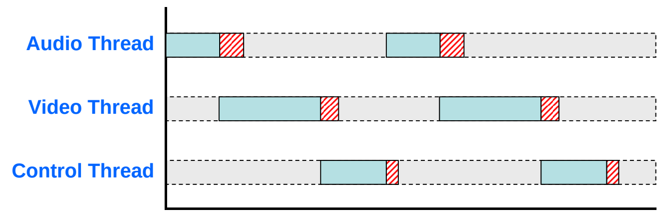
V4L2 driver: VIDIOC_DQBUF ioctl is blocking

FBDEV driver: FBIO_WAITFORVSYNC ioctl is blocking

Multi-Thread System

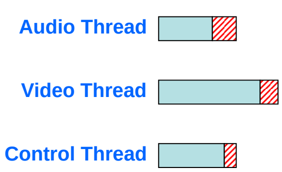
- Linux Processes
- Linux Threads
- Thread Synchronization
- Using Real-Time Threads
- Lab 08

Time-Sliced A/V Application, >100% load



- Adding a new thread of the highest “niceness” (smallest time slice) may disrupt lower “niceness” threads (higher time slices)
- All threads share the pain of overloading, no thread has time to complete all of its processing
- Niceness values may be reconfigured, but system unpredictability will often cause future problems
- In general, what happens when your system reaches 100% loading? Will it degrade in a well planned way? What can you do about it?

Time-Sliced A/V Application Analysis



Audio thread completes 80% of samples

Video thread drops 6 of 30 frames

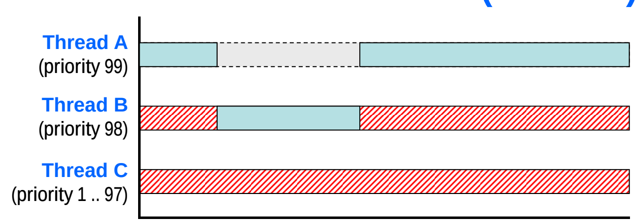
User response delayed 1mS

<u>All threads suffer, but not equally:</u>

- **Audio thread real-time failure is highly perceptible**
- **Video thread failure is slightly perceptible**
- **Control thread failure is not remotely perceptible**

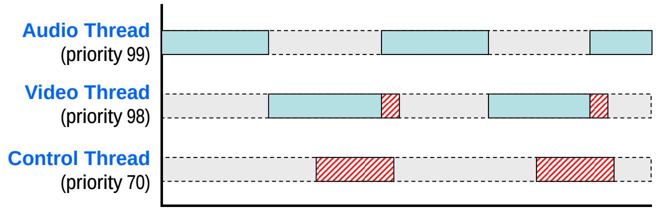
Note:
Time-slicing may also cause real-time failure in systems that are <100% loaded due to increased thread latency

Linux Real-Time Scheduler (Generic)



- In Linux, Real-Time threads are scheduled according to priority (levels 1-99, where time-slicing is effectively level 0)
- The highest priority thread always “wins” and will run 100% of the time unless it blocks

Real-time A/V Application, >100% load



- Audio thread is guaranteed the bandwidth it needs
- Video thread takes the rest
- Control thread never runs!

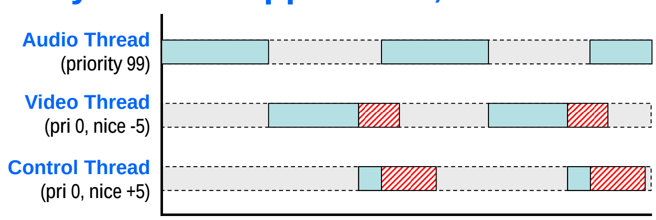
Time-Sliced A/V Application Analysis

Audio Thread	**Audio thread completes, no distortion**
Video Thread	**Video thread drops 1 of 30 frames**
Control Thread	**No user response**

Still a problem:

- **Audio thread completes as desired**
- **Video thread failure is practically inperceptible**
- **Control thread never runs – User input is locked out**

Hybrid A/V Application, >100% load



- Audio thread is guaranteed the bandwidth it needs
- Video thread takes *most* of remaining bandwidth
- Control thread gets a small portion of remaining bandwidth

Hybrid A/V Application Analysis

Audio Thread	**Audio thread completes, no distortion**
Video Thread	**Video thread drops 2 of 30 frames**
Control Thread	**User response delayed 100ms**

A good compromise:

- **Audio thread completes as desired**
- **Video thread failure is barely perceptible**
- **Control thread delayed response is acceptable**
- **Bottom Line: We have designed the system so that it degrades gracefully**

How do I choose the scheduling method?

Default Thread Scheduling

```
#include <pthread.h>
  …
pthread_create(&myThread, NULL, my_fxn,
                          (void *) &audio_env);
```

- Setting the second argument to NULL means the pthread is created with default attributes

pThread attributes:	NULL / default value:
stacksize	PTHREAD_STACK_MIN
…	…
detachedstate	PTHREAD_CREATE_JOINABLE
schedpolicy	SCHED_OTHER (time slicing)
inheritsched	PTHREAD_INHERIT_SCHED
schedparam.sched_priority	0

Scheduling Policy Options

	SCHED_OTHER	SCHED_RR	SCHED_FIFO
Sched Method	Time Slicing	Real-Time (RT)	
RT priority	0	1 to 99	1 to 99
Min niceness	+20	n/a	n/a
Max niceness	-20	n/a	n/a
Scope	root or user	root	root

- Time Sliced scheduling is specified with SCHED_OTHER:
  - Niceness determines how much time slice a thread receives, where higher niceness value means less time slice
  - Threads that block frequently are rewarded by Linux with lower niceness
- Real-time threads use preemptive (i.e. priority-based) scheduling
  - Higher priority threads always preempt lower priority threads
  - RT threads scheduled at the same priority are defined by their policy:
    - SCHED_FIFO: When it begins running, it will continue until it blocks
    - SCHED_RR: "Round-Robin" will share with other threads at it's priority based on a deterministic time quantum

Real-time Thread Creation Procedure

Create attribute structure

```
// Initialize the pthread_attr_t structure audioThreadAttrs
pthread_attr_init(&audioThreadAttrs);
```

Set attribute to real-time priority 99

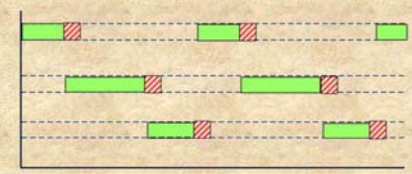
```
// Set the inheritance value in audioThreadAttrs structure
pthread_attr_setinheritsched(&audioThreadAttrs,
                             PTHREAD_EXPLICIT_SCHED);

// Set the scheduling policy for audioThreadAttrs structure
pthread_attr_setschedpolicy(&audioThreadAttrs, SCHED_RR);

// Set the scheduler priority via audioThreadParams struct
audioThreadParams.sched_priority = 99;
Pthread_attr_setschedparam(&audioThreadAttrs,
                           &audioThreadParams);
```

Create thread with given attributes

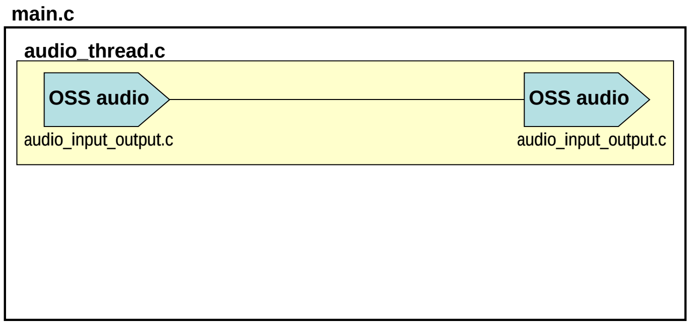
```
// Create the new thread using thread attributes
pthread_create(&audioThread, &audioThreadAttrs,
               audio_thread_fxn, (void *) &audio_env);
```

Multi-Thread System

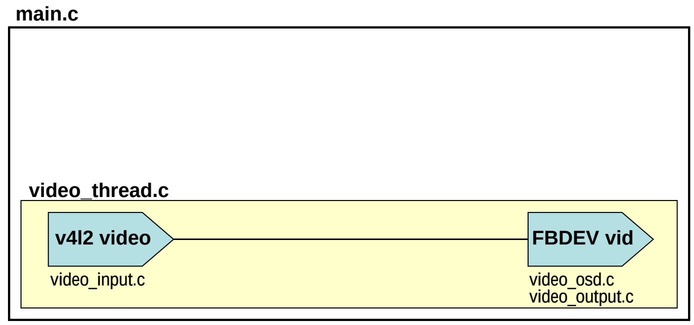
- Linux Processes
- Linux Threads
- Thread Synchronization
- Using Real-Time Threads
- Lab 08



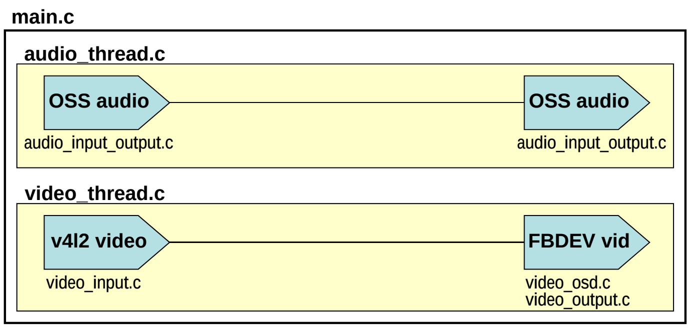
lab06c_audio_loopthru



lab07d_video_loopthru



lab08a_audio_video



Lab 8

1. In lab08b_audio_video_rtime, main.c, what priority is the audio thread set to? (You may use an expression here.)

2. When running the debug version of the application (./app_debug.x470MV), what does the debug output indicate the audio thread priority is set to? (Numerical value)

3. What priority is the video thread set to? (You may use an expression here.)

4. When running the debug version of the application (./app_debug.x470MV), what does the debug output indicate the video thread priority is set to? (Numerical value)